Optimizing cloud applications with DB2 stored procedures

Vaibhav Kulkarni (vaibhavk@in.ibm.com)
Performance Analyst
IBM

Madhuri Vuddaraju (madhuri.v@in.ibm.com)
Staff Engineer
IBM

12 February 2015

This tutorial describes the IBM® DB2® stored procedure framework, methods to monitor stored procedure performance, and methods to optimize stored procedure performance. DB2 provides a routine monitoring framework that helps pinpoint the statements or parts of the procedure code that can be tuned for better performance. The tutorial also describes good practices for writing DB2 SQL/PL and Oracle PL/SQL procedures and simple way of migrating Oracle PL/SQL procedures to DB2.

As more applications are written for the cloud environment, the focus for developers is to push as much functionality as possible to the database server. This is where stored procedures come into play. While stored procedures are effective for concentrating the application logic on the server side and helping to reduce client/server network communication, poorly written stored procedures can offset the gains obtained from reducing the network traffic. From an application developer perspective, it is important to know and use the methods and techniques for monitoring and optimizing stored procedures to achieve tangible gains in cloud application performance.

In DB2, stored procedures are a type of routine. In this tutorial, the terms routine and stored procedure are used interchangeably.

IBM DB2 stored procedure framework

DB2 provides a framework for executing built-in and user-defined routines. The framework is transparent to the user, but the user can control some aspects of it. All routines are executed in a db2fmp process, which is executed under the user specified when the instance is created. By executing the routines in a separate address space, DB2 can isolate the execution and logic from the database manager and prevent any harmful procedures from corrupting the DB2 address space.
DB2 uses the following parameters to control certain behaviors of how stored procedures are executed:

**KEEPFENCED**
This database configuration parameter indicates whether a `db2fmp` process is retained after a fenced mode routine call is complete. If `KEEPFENCED` is set to `YES`, the `db2fmp` process is reused for subsequent procedure calls. By reusing the fenced mode process, it reduces the time it takes to initialize the process and can help small procedures where the initialization cost is significant. Set this to `NO` only if your procedures are thread-unsafe procedures.

**FENCED_POOL**
This database configuration parameter specifies the number of `db2fmp` processes that are cached. If the application frequently invokes routines, creating and destroying the `db2fmp` process can impact performance negatively. By using this parameter, you can avoid creating and destroying the framework for every procedure call.

**NUM_INITFENCED**
This parameter specifies the number of `db2fmp` processes to be started when instances are started. These processes are common processes for all the databases under the instance. By starting only a few `db2fmp` processes when the instance is started, you can reduce the impact of starting and initializing the processes when the first few routines are executed.

Other procedure clauses you need to be familiar with:

**FENCED THREADSAFE clause**
Include this clause in a `CREATE PROCEDURE` statement. Procedures that are created with the `FENCED THREADSAFE` clause run in the same process as other routines. Routines written in languages other than Java® share one process, while Java routines share another process. This separation protects Java routines from the potentially more error-prone routines that are written in other languages. Also, the process for Java routines contains a Java VM, which incurs a high memory cost and is not used by other routine types. Multiple invocations of `FENCED THREADSAFE` routines share resources and therefore incur less system impact.

**FENCED NOT THREADSAFE clause**
Include this clause in a `CREATE PROCEDURE` statement. Procedures created with this clause, execute in their own dedicated process. If you are running numerous routines, this setup can have a detrimental effect on database system performance. Use the `FENCED NOT THREADSAFE` clause when you register the routine only if the routine is not safe enough to run in the same process as other routines. Otherwise, use the `FENCED THREADSAFE` clause for better performance.

**Monitoring Stored Procedure performance**
This section introduces routine monitoring, related parameters, and table functions that can be leveraged to monitor stored procedures, and other user-defined functions.

The routine monitoring feature was introduced in DB2 V10.1 Fix Pack 2 as a couple of database configuration parameters and monitoring table functions, which are described in this tutorial. Routine monitoring can:
• Identify slow running stored procedures and functions. It can identify bottle necks that cause performance of routines to degrade.
• Provide constant monitoring, enabling users to get real-time performance metrics for stored procedures and other routines.
• Be enabled or disabled at the database level and affect any routine invocations within the database.
• Be configured online. Routine performance information is stored in the database heap.

Configuration parameters for enabling routine monitoring

Table 1 lists the database configuration parameters that you can use to enable routine monitoring.

### Table 1. Database configuration parameters to enable routine monitoring

<table>
<thead>
<tr>
<th>Database configuration parameter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON_RTN_DATA</td>
<td>NONE</td>
<td>Set to BASE to enable monitoring of routines. Information about the routines that are executed is collected in the database heap and can be accessed by using the table functions MON_GET_ROUTINE and MON_GET_ROUTINE_DETAILS.</td>
</tr>
<tr>
<td>MON_RTN_EXECLIST</td>
<td>OFF</td>
<td>Set to ON to enable monitoring of routines at the statement level when MON_RTN_DATA is set to BASE. Monitoring data can be accessed by querying table functions with MON_GET_RTN_EXEC_LIST.</td>
</tr>
</tbody>
</table>

Routine Monitoring Table Functions

Table functions MON_GET_ROUTINE and MON_GET_RTN_EXEC_LIST can be queried to obtain aggregated performance metrics for routines.

**MON_GET_ROUTINE table function**

MON_GET_ROUTINE helps identify the most expensive routines on the database server. It returns one row of performance data for each of the routines that satisfies the input parameters. Metrics that are returned are aggregations of all the executions of a routine; that is, if a routine is executed more than once, then the table function returns metrics that are aggregated for all the executions.

The syntax for the MON_GET_ROUTINE table function is:

```sql
MON_GET_ROUTINE(<routine_type>, <routine_schema>, <routine_module>, <routine_name>, <member>)
```

**routine_type**

An input parameter of type CHAR(2) that specifies the type of routine or compiled trigger for which data is returned:
• "P" for a procedure
• "SP" for the specific name of a procedure
• "F" for a compiled function
• "SF" for the specific name of a compiled function
• "T" for a compiled trigger
• "C" for a dynamically prepared compound SQL statement or an anonymous block in PL/SQL

**routine_schema**

An input parameter of type VARCHAR(128) that specifies the schema for the routine or trigger. For dynamically prepared compound SQL statements or anonymous blocks, the schema can be determined by using the MON_GET_SECTION_ROUTINES table function. Use NULL or an empty string to return the routines and triggers in all schemas. This parameter is case-sensitive.

**routine_module_name**

An input parameter of type VARCHAR(128) that specifies the name of the module for the input routine, if applicable. Use NULL or an empty string to return the routines in all modules. This parameter is case-sensitive.

**routine_name**

An input parameter of type VARCHAR(128) that specifies the name of the routine. If the input parameter is "SP" or "SF", the specific name of the routine must be provided. For dynamically prepared compound SQL statements or anonymous blocks, the name can be determined using the MON_GET_SECTION_ROUTINES table function. Use NULL or an empty string to return all routines that match the other input parameters. This parameter is case-sensitive.

**member**

An input parameter of type INTEGER that specifies a valid member in the same instance as the currently connected database when you call this function. Specify -1 for the current database member, or -2 for all database members. If the NULL value is specified, -1 is implicitly set.

**Note:** Input parameters are complimentary and a NULL can be specified to obtain the metrics for all the routines that were run against the database.

**MON_GET_ROUTINEDETAILS** tab function

**MON_GET_ROUTINEDETAILS** returns routine metrics in an XML document format.

The syntax to use this table function is similar to the **MON_GET_ROUTINE** table function.

**MON_GET_RTN_EXEC_LIST** table function

**MON_GET_RTN_EXEC_LIST** returns the aggregated performance metrics for the list of statements that are executed by a routine. This function also returns metrics that are aggregated across all executions of that statement within that routine. Each statement in the output is identified by an executable ID which can be used with **MON_GET_PKG_CACHE_STMT** to obtain the statement information.

The syntax for the **MON_GET_RTN_EXEC_LIST** table function is:

```
MON_GET_RTN_EXEC_LIST(<routine_type>, <routine_schema>, <routine_module>, <routine_name>, <member>)
```
Input parameters that are accepted by `MON_GET_RTN_EXEC_LIST` are the same as those of the `MON_GET_ROUTINE` table function.

Some useful metrics like `TOTAL_ROUTINE_TIME`, `LOCK_WAIT_TIME`, `CPU_TIME`, `ROWS_READ`, `ROWS_RETURNED` can be queried by using these table functions.

For more information about monitoring table functions and the metrics that can be queried, see the DB2 documentation in IBM Knowledge Center.

**Note:** Execution metrics data is deleted when the database configuration parameter `MON_RTN_DATA` is disabled or when the database is deactivated.

### Identifying performance issues with procedures.

Now, let's look at a simple example of identifying and resolving stored procedure performance issue by querying table functions.

**Listing 1** shows the list of all the executed routines ordered by the CPU consumption time. `MON_GET_ROUTINE` returns other performance metrics as well for each routine. However, for simplicity we are querying only `TOTAL_CPU_TIME` in this example. For more information about fields in the routine monitoring table functions, see the DB2 documentation in IBM Knowledge Center.

**Listing 1. Query to list executed routines for a given schema ordered by execution time**

```sql
select char(ROUTINE_TYPE,15) as ROUTINE_TYPE, char(ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA ,
char(ROUTINE_NAME,15) as ROUTINE_NAME , TOTAL_ROUTINE_TIME,(TOTAL_CPU_TIME/TOTAL_TIMES_ROUTINE_INVOKED) AS CPU_TIME_AVG ,TOTAL_TIMES_ROUTINE_INVOKED as NUM_INVOCATIONS from TABLE(MON_GET_ROUTINE(NULL, 'TPCE', NULL, NULL, -1)) order by TOTAL_ROUTINE_TIME DESC
```

**Listing 2. Results of the previous query**

<table>
<thead>
<tr>
<th>ROUTINE_TYPE</th>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ROUTINE_TIME</th>
<th>CPU_TIME_AVG</th>
<th>NUM_INVOCATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>TPCE</td>
<td>BROKERVOLUME_F1</td>
<td>3483</td>
<td>34051</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>TPCE</td>
<td>MARKETWATCH_F1</td>
<td>1417</td>
<td>8790</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>TPCE</td>
<td>MARKETFEED_F1</td>
<td>736</td>
<td>975</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Schema name `TPCE` is supplied as an input parameter to the table function `MON_GET_ROUTINE` to filter routine information for schema `TPCE`.

Routine `BROKERVOLUME_F1` takes the highest execution time. To obtain the statement level execution information for this routine, run the queries `MON_GET_RTN_EXEC_LIST` and `MON_GET_PKG_CACHE_STMT` (see Listing 3 ).
Listing 3. Query to list the statements executed by a routine ordered by execution time

```
select  char(B.ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(B.ROUTINE_NAME,15) as ROUTINE_NAME ,char(C.STMT_TEXT,50) as STMT_TEXT ,(B.TOTAL_CPU_TIME/B.NUM_EXECUTIONS) as CPU_TIME, B.NUM_EXECUTIONS from TABLE(MON_GET_ROUTINE_EXEC_LIST(NULL, 'TPCE', NULL , 'BROKERVOLUME_F1', -1)) AS B, TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) AS C where B.executable_id=C.executable_id order by CPU_TIME DESC
```

Listing 4. Results of previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>STMT_TEXT</th>
<th>CPU_TIME</th>
<th>NUM_EXECUTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-----------</td>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>TPCE</td>
<td>BROKERVOLUME_F1</td>
<td>SELECT ARRAY_AGG(B_NAME), ARRAY_AGG(CAST(VOLUME AS DECIMAL(1)</td>
<td>34012</td>
<td>100</td>
</tr>
</tbody>
</table>

```
SQL0445W Value "SELECT ARRAY_AGG(B_NAME), ARRAY_AGG(CAST(VOLUME AS DECIMAL(1" has been truncated. SQLSTATE=01004
```

**Note:** Schema and routine names are supplied as input parameters to the table function `MON_GET_ROUTINE_EXEC_LIST`.

BROKERVOLUME_F1 is executing only one `SELECT` statement, which is mounting to the higher overall execution time. Next, run a query to obtain the rows read and returned by this SQL statement (see Listing 5).

Listing 5. Query to list rows read and returned by each statement in a routine

```
select  char(B.ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(B.ROUTINE_NAME,15) as ROUTINE_NAME, char(C.STMT_TEXT,10) as STMT_TEXT,(C.ROWS_READ/C.NUM_EXECUTIONS) as ROWS_READ, (C.ROWS_RETURNED/C.NUM_EXECUTIONS) as ROWS_RETURNED,B.NUM_EXECUTIONS from TABLE(MON_GET_ROUTINE_EXEC_LIST(NULL, 'TPCE', NULL, 'BROKERVOLUME_F1', -1)) AS B, TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) as C where B.executable_id=C.executable_id
```

Listing 6. Results of previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>STMT_TEXT</th>
<th>ROWS_READ</th>
<th>ROWS_RETURNED</th>
<th>NUM_EXECUTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>TPCE</td>
<td>BROKERVOLUME_F1</td>
<td>SELECT ARR</td>
<td>67850</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

```
SQL0445W Value "SELECT ARRAY_AGG(B_NAME), ARRAY_AGG(CAST(VOLUME AS DECIMAL(1" has been truncated. SQLSTATE=01004
```

**Note:** Schema and routine names are supplied as input parameters to the table function `MON_GET_ROUTINE_EXEC_LIST`.

The difference between the number of `ROWS_READ` and `RETURNED` suggests that the query is reading lot of rows to return just a few rows. This result suggests that there might be a performance issue with the query. The query is most likely spending time in scanning and filtering rows. To confirm the issue, you can obtain the access plan of this SQL statement (see Listing 7).
Listing 7. SELECT statement being executed by routine

```sql
SELECT ARRAY_AGG(B_NAME), ARRAY_AGG(CAST(VOLUME AS DECIMAL(12,2))), COUNT(B_NAME)
INTO :HV00011, :HI00011, :HV00012, :HI00012, :HV00010, :HI00010
FROM ( SELECT B_NAME, SUM(TR_QTY * TR_BID_PRICE) AS VOLUME
FROM BROKER,
UNNEST(CAST(:HV00008, :HI00008 AS "TPCE"."CHAR49_ARR_40") AS B_LIST(NAME),
COMPANY, TRADE_REQUEST,
SECURITY, SECTOR, INDUSTRY
WHERE B_NAME = B_LIST.NAME
AND TR_S_SYMB = S_SYMB
AND TR_B_ID = B_ID
AND S_CO_ID = CO_ID
AND CO_IN_ID = IN_ID
AND SC_ID = IN_SC_ID
AND SC_NAME = :HV00009, :HI00009
GROUP BY B_NAME
ORDER BY VOLUME DESC
)
```

To obtain the access plan of a routine statement, follow these steps.

**Step 1.** Obtain the executable ID of the previous SQL statement in Listing 7.

```sql
SELECT executable_id FROM TABLE(MON_GET_ROUTINE_EXEC_LIST(NULL, 'TPCE', NULL, 'BROKERVOLUME_F1', -1))
```

The results of this query are:

<table>
<thead>
<tr>
<th>EXECUTABLE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>x'01000000000000003902000000000001000000010020141113050649093556'</td>
</tr>
</tbody>
</table>

1 record(s) selected.

**Step 2.** With the executable_id returned in the previous step, make the following call:

```sql
call EXPLAIN_FROM_SECTION(x'01000000000000003902000000000001000000010020141113050649093556',
'M', NULL, 'TPCE', ?, ?, ?, ?, ?)
```

**Step 3.** Run this command to extract the access plan:

```
db2exfmt -d <db name> -g TIC -w -1 -n % -s % -# 0 -o exfmt.out
```

**Step 4.** Review the access plan:

```
Access Plan
-----------
Total Cost:             900.218
Query Degree:           1
```

<table>
<thead>
<tr>
<th>Rows RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
A closer look at the access plan reveals that access to table TRADE_REQUEST can be optimized.
A full tablescan is being performed on TRADE_REQUEST. An index can be created to replace TBSCAN(Operator : 12) with IXSCAN. Create an index on TRADE_REQUEST table containing columns accessed by the query.

Columns TR_S_SYMB, TR_B_ID from table TRADE_REQUEST are being used in the WHERE clause. While the columns TR_QTY and TR_BID_PRICE are being used in the SELECT clause. To ensure that the query uses just the IXSCAN to obtain all the required fields of the table, create an INDEX on (TR_S_SYMB, TR_B_ID) and include columns (TR_BID_PRICE, TR_QTY) as part of the index.

```
CREATE UNIQUE INDEX "TPCE  "."TR_IDX1" ON "TPCE  "."TRADE_REQUEST" ("TR_B_ID" ASC,  "TR_S_SYMB" ASC,  "TR_T_ID" ASC) INCLUDE ("TR_BID_PRICE",  "TR_QTY")  PCTFREE 0 LEVEL2 PCTFREE 0 COMPRESS NO INCLUDE NULL KEYS ALLOW REVERSE SCANS
```

Also, the IXSCAN(Operator : 14) and FETCH(Operator : 13) on table BROKER can be optimized by creating an INDEX that includes all the columns in table BROKER that are referenced by the query. Columns B_NAME and B_ID from table BROKER are being referenced in the query. So, you create an INDEX on columns B_NAME and B_ID.

```
CREATE INDEX "TPCE  "."B_IDX1" ON "TPCE  "."BROKER" ("B_NAME" ASC,  "B_ID" ASC) PCTFREE 0 LEVEL2 PCTFREE 0 COMPRESS NO INCLUDE NULL KEYS ALLOW REVERSE SCANS
```
Recompile the routine using either the REBIND command or by re-creating the routine. To use the REBIND command:

"CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'TPCE.BROKERVOLUME_F1','ANY')"

Or, re-create the routine:

"CREATE OR REPLACE PROCEDURE BROKERVOLUME_F1" ...procedure definition.

Now lets query MON_GET_ROUTINE to list the executed routines for a given schema ordered by execution time.

**Listing 8. Rerun the query to list executed routines for a given schema ordered by execution time**

```sql
select char(ROUTINE_TYPE,15) as ROUTINE_TYPE, char(ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA,
char(ROUTINE_NAME,15) as ROUTINE_NAME,
TOTAL_ROUTINE_TIME,(TOTAL_CPU_TIME/TOTAL_TIMES_ROUTINE_INVOKED) AS CPU_TIME_AVG,
TOTAL_TIMES_ROUTINE_INVOKED as NUM_INVOCATIONS from TABLE(MON_GET_ROUTINE(NULL, 'TPCE', NULL, NULL, -1)) order by TOTAL_ROUTINE_TIME DESC
```

**Listing 9. Results of the previous query**

<table>
<thead>
<tr>
<th>ROUTINE_TYPE</th>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ROUTINE_TIME</th>
<th>CPU_TIME_AVG</th>
<th>NUM_INVOCATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>TPCE</td>
<td>MARKETWATCH_F1</td>
<td>2102</td>
<td>5644</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>TPCE</td>
<td>BROKERVOLUME_F1</td>
<td>1337</td>
<td>13309</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>TPCE</td>
<td>MARKETFEED_F1</td>
<td>166</td>
<td>908</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Schema name TPCE is supplied as an input parameter to the table function MON_GET_ROUTINE to filter the routine information for schema TPCE.

By using these queries and updating the routines, the execution time of the routine BROKERVOLUME_F1 has improved significantly. **Listing 10** shows the access plan after creating the index:

**Listing 10. Access plan of the statement after creation of indices**

```
Access Plan:  
-------------
Total Cost: 745.603
Query Degree: 1

Rows
RETURN
( 1)
Cost
I/O
 1
GRPBY
( 2)
745.603
NA
```
The index TR_ID1X is being accessed to obtain data from the TRADEDETAILS table, and
B_ID1X is being accessed to obtain data from the BROKER table.

**Monitoring stored procedures and functions using db2pd**

The db2pd tool provides a couple of options to monitor the db2fmp processes and the routines
executed in a limited way. This tool can be useful for quick monitoring.

The `db2pd -fmp` option shows the procedures that are currently executing.

**Listing 11. Output from running the `db2pd -fmp` option**

$ db2pd -fmp
Database Member 0 -- Active -- Up 10 days 12:33:30 -- Date 2014-11-30-12.16.49.116319
FMP:
Pool Size: 2
Max Pool Size: 200 (Automatic)
Keep FMP: YES
Initialized: YES
Trusted Path: /home/db2inst1/sqlib/function/unfenced
Fenced User: db2fence
Shared Memory: 0x000000002011C020
IPC Pool: 0x000000002011C040

FMP Process:
Address FmpPid Bit Flags ActiveThrd PooledThrd ForcedThrd Active IPCList
0x00000002021C7F40 15684 64 0x00000000 0 0 0 No 0x00000002021C89E0

Active Threads:
Address FmpPid EduPid ThreadId
No active threads.

Pooled Threads:
Address FmpPid ThreadId
No pooled threads.

Forced Threads:
Address FmpPid ThreadId
No forced threads.
The db2pd -fmpe option shows the history of a particular process and the procedures that it has executed.

**Listing 12. The output from running the db2pd -fmpe option**

```bash
$ db2pd -fmpe pid=15684
```

Database Member 0 -- Active -- Up 10 days 12:39:59 -- Date 2014-11-30-12.23.18.398800

FMP Process:

<table>
<thead>
<tr>
<th>FmpPid</th>
<th>Bit</th>
<th>Flags</th>
<th>ActiveThrd</th>
<th>PooledThrd</th>
<th>ForcedThrd</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>15684</td>
<td>64</td>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No</td>
</tr>
</tbody>
</table>

Active Threads:
No active threads.

Pooled Threads:
ThreadID : 0
Routine ID    Timestamp
65703  2014-11-30-12.18.22.938832

Forced Threads:
No forced threads.

The db2pd -fmpe genquery option outputs the query that can be executed to find out the procedure name that was executed earlier.

**Listing 13. Output from running the db2pd -fmpe_genquery option**

```bash
$ db2pd -fmpe pid=15684 n=1 genquery
```

Database Member 0 -- Active -- Up 10 days 12:43:10 -- Date 2014-11-30-12.26.29.707007

```sql
WITH RTNHIST ( PID, TID, RTNID, RTNTIME) AS
( VALUES   ( 15684     , 0         , 65703     , TIMESTAMP('2014-11-30-12.18.22.938832'))
)
SELECT R.PID, R.TID, R.RTNTIME, ROUTINESCHEMA, ROUTINENAME, SPECIFICNAME, ROUTINEID
FROM syscat.routines, RTNHIST as R
where ROUTINEID = R.RTNID
ORDER BY R.PID, R.TID, R.RTNTIME
```

**Monitoring PL/SQL procedures**

DB2 provides an *Oracle compatibility feature* that reduces the time and complexity of enabling applications written for Oracle databases on DB2. The feature can be enabled by setting a registry environment variable DB2_COMPATIBILITY_VECTOR=ORA. One important part of this feature is the ability to execute Oracle PL/SQL procedures and packages (not be to be confused with DB2 packages) on DB2 without major changes. Like DB2 stored procedures, performance of PL/SQL procedures can also be monitored using the routine monitoring feature described in the previous section.

**Static stored procedures**

It is very common to have nested PL/SQL procedures where one procedure invokes another procedure and so on. Procedures do provide the flexibility of separating the logic and modularizing the code. But, one common problem observed in such nested PL/SQL procedures, especially simple procedures, is the relatively high cost that is associated with executing the procedures. In most cases, the stored procedure framework cost is negligible. But, in cases where the procedure
logic itself is very small or the procedure has only one or two simple queries, the time taken to execute the framework-related code can be significant compared to the execution time of the queries and logic inside the procedure. Having too small procedures can negate the performance benefits of having procedures in the first place. If creating smaller procedures is unavoidable, using the static procedure can help reduce the framework cost.

Let's look at the performance metrics of one such PL/SQL routines to identify the cause for excess time consumption.

Consider a simple PL/SQL package which has four procedures executing a simple `SELECT` statement each. All the four procedures are nested as can be seen in the example below. Procedure P1 is called by the user with PINCODE as the input to retrieve City, State, Country and Continent Names.

**Listing 14. Sample PL/SQL static stored procedure**

```sql
create or replace package Package1 as

procedure P1(
  pincode in INT DEFAULT 560069,
  City OUT CHAR(30),
  State OUT CHAR(30),
  Country OUT CHAR(30),
  Continent OUT CHAR(30)) ;

procedure P2
  (City IN CHAR(30),
   State OUT CHAR(30),
   Country OUT CHAR(30),
   Continent OUT CHAR(30)) ;

procedure P3
  (State IN CHAR(30),
   Country OUT CHAR(30),
   Continent OUT CHAR(30)) ;

procedure P4
  (Country IN CHAR(30),
   Continent OUT CHAR(30)) ;

end Package1;
@

create or replace package body Package1 as

procedure P1(
  Pincode in INT DEFAULT 560069,
  City OUT CHAR(30),
  State OUT CHAR(30),
  Country OUT CHAR(30),
  Continent OUT CHAR(30)) as
  BEGIN
  Select city into City from Table_city where pincode=Pincode;
  P2(City,State,Country,Continent);
  END P1;
```
procedure P2(
    City IN CHAR(30),
    State OUT CHAR(30),
    Country OUT CHAR(30),
    Continent OUT CHAR(30)) as
BEGIN
    Select state into State from Table_state where city=City;
    P3(State,Country,Continent);
END P2;

procedure P3(
    State IN CHAR(30),
    Country OUT CHAR(30),
    Continent OUT CHAR(30)) as
BEGIN
    Select country into Country from Table_country where state=State;
    P4(Country,Continent);
END P3;

procedure P4
    (Country IN CHAR(30),
    Continent OUT CHAR(30)) as
BEGIN
    Select Continent into Continent from Table_continent where country=Country;
END P4;
end Package1
@

Enable monitoring and execute the above procedure in a loop to gather performance metrics.

**Step 1.** Call the package

call package1.P1(560069,?,?,?,?)

**Listing 15. Results of the previous statement**

<table>
<thead>
<tr>
<th>Value of output parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Name</td>
</tr>
<tr>
<td>Parameter Value</td>
</tr>
<tr>
<td>Parameter Name</td>
</tr>
<tr>
<td>Parameter Value</td>
</tr>
<tr>
<td>Parameter Name</td>
</tr>
<tr>
<td>Parameter Value</td>
</tr>
<tr>
<td>Parameter Name</td>
</tr>
<tr>
<td>Parameter Value</td>
</tr>
<tr>
<td>Return Status</td>
</tr>
</tbody>
</table>

**Step 2.** Execute the above procedure in a loop to gather performance metrics.

**Step 3.** Gather metrics with **MON_GET_ROUTINE**.
Listing 16. Query using MON_GET_ROUTINE

```
select char(ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(ROUTINE_NAME,15) as ROUTINE_NAME ,
     TOTAL_ROUTINE_TIME , TOTAL_CPU_TIME , TOTAL_TIMES_ROUTINE_INVOKED from TABLE(MON_GET_ROUTINE(NULL, 'TPCE',
     NULL, NULL, -1))  order by TOTAL_ROUTINE_TIME DESC
```

Listing 17. Results of the previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ROUTINE_TIME</th>
<th>TOTAL_CPU_TIME</th>
<th>TOTAL_TIMES_ROUTINE_INVOKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCE</td>
<td>P1</td>
<td>54</td>
<td>310998</td>
<td>100</td>
</tr>
<tr>
<td>TPCE</td>
<td>P2</td>
<td>35</td>
<td>168350</td>
<td>100</td>
</tr>
<tr>
<td>TPCE</td>
<td>P3</td>
<td>21</td>
<td>39846</td>
<td>100</td>
</tr>
<tr>
<td>TPCE</td>
<td>P4</td>
<td>8</td>
<td>9357</td>
<td>100</td>
</tr>
</tbody>
</table>

4 record(s) selected.

Step 4. Gather metrics with MON_GET_ROUTINE_EXEC_LIST.

Listing 18. Query with MON_GET_ROUTINE_EXEC_LIST

```
select  char(B.ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(B.ROUTINE_NAME,15) as ROUTINE_NAME ,
        B.TOTAL_ACT_TIME, char(C.STMT_TEXT,50) as STMT_TEXT from  TABLE(MON_GET_ROUTINE_EXEC_LIST(NULL,
        'TPCE', NULL , NULL, -1)) AS B, TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) AS C where
        B.executable_id=C.executable_id
```

Listing 19. Results from the previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ACT_TIME</th>
<th>STMT_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCE</td>
<td>P1</td>
<td>7</td>
<td>SELECT CITY INTO :HV00009 :HI00009 FROM TABLE_CITY where P&quot; has been truncated. SQLSTATE=01004</td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>9</td>
<td>CALL P2(:HV00009 :HI00009 , :HV00010 :HI00010 , &quot; has been truncated. SQLSTATE=01004</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>12</td>
<td>CALL P3(:HV00009 :HI00009 , :HV00010 :HI00010 , &quot; has been truncated. SQLSTATE=01004</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>2</td>
<td>SELECT STATE INTO :HV00009 :HI00009 FROM TABLE_STATE where&quot; has been truncated. SQLSTATE=01004</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>2</td>
<td>SELECT COUNTRY INTO :HV00009 :HI00009 FROM TABLE_COUNTRY w&quot; has been truncated. SQLSTATE=01004</td>
</tr>
</tbody>
</table>
The amount of time for the statements to execute in the routine is considerably low, in comparison with the overall routine execution time. To overcome the performance impact with nested procedures, we can adopt static procedures. Procedure P2 definition can be embedded into procedure P1 as shown in Listing 20

### Listing 20. Embedded procedures

```sql
create or replace package body Package1 as
procedure P1(
P in INT DEFAULT 560069,
City OUT CHAR(30),
State OUT CHAR(30),
Country OUT CHAR(30),
Continent OUT CHAR(30)) as
    procedure P2(
        City IN CHAR(30),
        State OUT CHAR(30),
        Country OUT CHAR(30),
        Continent OUT CHAR(30)) as
        procedure P3(
            State IN CHAR(30),
            Country OUT CHAR(30),
            Continent OUT CHAR(30)) as
            procedure P4(
                Country IN CHAR(30),
                Continent OUT CHAR(30)) as
            BEGIN
            Select Continent into Continent from Table_continent where country=Country;
            END P4;
            BEGIN
            Select country into Country from Table_country where state=State;
            P4(Country,Continent);
            END P3;
            BEGIN
            Select state into State from Table_state where city=City;
            P3(State,Country,Continent);
            END P2;
            BEGIN
            Select city into City from Table_city where pincode=P;
            P2(City,State,Country,Continent);
            END P1;
    END P2;
    BEGIN
    Select Continent into Continent from Table_continent where country=Country;
    END P4;
    BEGIN
    Select country into Country from Table_country where state=State;
    P4(Country,Continent);
    END P3;
    BEGIN
    Select state into State from Table_state where city=City;
    P3(State,Country,Continent);
    END P2;
    BEGIN
    Select city into City from Table_city where pincode=P;
    P2(City,State,Country,Continent);
    END P1;
```

Run the above procedure in a loop and obtain the performance metrics for this routine

### Listing 21. Performance of stored procedure with embedded procedures

```sql
select char(ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(ROUTINE_NAME,15) as
ROUTINE_NAME ,TOTAL_ROUTINE_TIME , TOTAL_CPU_TIME , TOTAL_TIMES_ROUTINE_INVOKED from
TABLE(MON_GET_ROUTINE(NULL, 'TPCE', NULL, NULL, -1)) order by TOTAL_ROUTINE_TIME DESC
```
Listing 22. Results of the previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ROUTINE_TIME</th>
<th>TOTAL_CPU_TIME</th>
<th>TOTAL_TIMES_ROUTINE_INVOKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCE</td>
<td>P1</td>
<td>64</td>
<td>109074</td>
<td>100</td>
</tr>
</tbody>
</table>

Listing 23. Performance of individual statements with embedded procedures

```sql
select char(B.ROUTINE_SCHEMA,15) as ROUTINE_SCHEMA , char(B.ROUTINE_NAME,15) as ROUTINE_NAME , B.TOTAL_ACT_TIME, char(C.STMT_TEXT,50) as STMT_TEXT from  TABLE(MON_GET_ROUTINE_EXEC_LIST(NULL, 'TPCE', NULL, NULL, -1)) AS B, TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) AS C where B.executable_id=C.executable_id
```

Listing 24. Results of the previous query

<table>
<thead>
<tr>
<th>ROUTINE_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>TOTAL_ACT_TIME</th>
<th>STMT_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCE</td>
<td>P1</td>
<td>7</td>
<td>SELECT CITY INTO :HV00009  :HI00009  FROM TABLE_CITY where P&quot; has been truncated. SQLSTATE=01004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>CALL SYSIBMSUBROUTINE.P1_66720_2104221767(:HV00009  :HI00009)</td>
</tr>
</tbody>
</table>

2 record(s) selected with 2 warning messages printed.

The execution time of the stored procedure is significantly reduced with embedded static procedures, especially for smaller procedures and routines. This tutorial provides a Perl script that automates the embedding of procedures as static procedures. It accepts a routine name, routine schema, routine module, and database name as input and then embeds the input routine against all the parent routines that are calling the input routine with in the given package.

**Migrating Oracle PL/SQL procedures**

Most of the Oracle PL/SQL procedures should work without any changes on DB2 if you use the Oracle compatibility registry. IBM Data Studio can be a very useful tool when migrating data from an Oracle Source to a DB2 database. The Database Conversion Workbench plug-in is very useful in migrating Oracle PL/SQL statements and procedures to DB2.

IBM Database Conversion Workbench can analyze Oracle DDL and PL/SQL statements and create an assessment report that outlines the compatibility of the source Oracle database with DB2. The compatibility evaluation report includes an executive summary and a detailed list of DDL and PL/SQL incompatibilities, and some suggested workarounds to fix the incompatible code. The Database Conversion Workbench can help move database objects, PL/SQL routines from Oracle source to DB2 database. The Database Conversion Workbench also provides a conversion tool that can automatically convert certain known Oracle syntax incompatibilities to DB2 compatible syntax.
For more information about the conversion process, see the IBM Redbooks Oracle to DB2 Conversion Guide.

**Conclusion**

The routine monitoring framework makes it easy for developers and DBAs to figure out poorly performing stored procedures. Similarly, it helps identify the statements and queries within the routines that perform poorly. Corrective actions as shown in this tutorial can help improve stored procedure performance. The routine monitoring also works for PL/SQL procedures that were migrated from Oracle.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl script for inlining routines</td>
<td>inlineroutines.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Resources

Learn

• Get detailed information on migrating oracle PL/SQL procedures to DB2 in the IBM Redbooks Oracle to DB2 Conversion Guide: Compatibility Made Easy
• Refer to the DB2 documentation in IBM Knowledge Center for details on routine monitoring and other monitoring table functions
• Follow developerWorks on Twitter.

Get products and technologies

• Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

• Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
About the authors

Vaibhav Kulkarni

Vaibhav Kulkarni is an Advisory Software Engineer in DB2 for Linux, UNIX, and Windows. He works in the DB2 performance team. He provides technical support on customers’ performance problems.

Madhuri Vuddaraju

Madhuri Vuddaraju is an IBM Certified database administrator and works in the DB2 FVT team. Madhuri has been testing various aspects of column organized tables and other analytic features in DB2.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)